Remove All The Callbacks – BlackByte Ransomware Disables EDR Via RTCore64.sys Abuse

news.sophos.com/en-us/2022/10/04/blackbyte-ransomware-returns/

Andreas Klopsch October 4, 2022

With reports of a new data-leak site published by actors behind the BlackByte ransomware, we decided to take another look at the most recent variant written in Go.

We found a sophisticated technique to bypass security products by abusing a known vulnerability in the legitimate vulnerable driver RTCore64.sys. The evasion technique supports disabling a whopping list of over 1,000 drivers on which security products rely to provide protection. Sophos products provide mitigations against the tactics discussed in this article.

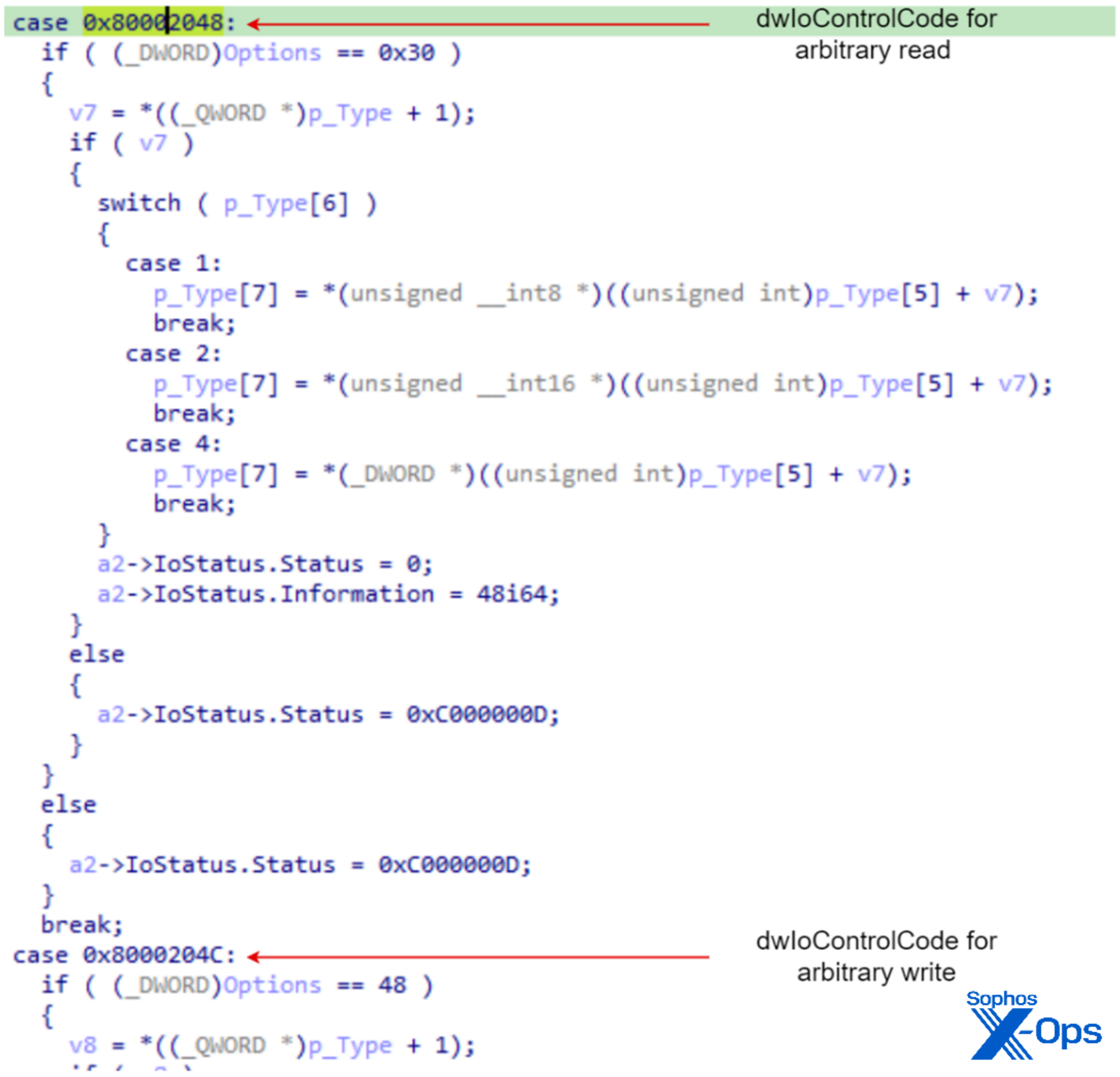
“Bring Your Own [Vulnerable] Driver” is the name given to this technique — exploiting a targeted system by abusing a legitimate signed driver with an exploitable vulnerability. In July 2022, Trend Micro reported on the abuse of a vulnerable anti-cheat driver for the game Genshin Impact, named mhyprot2.sys, to kill antivirus processes and services for mass-deploying ransomware. In May 2022, another report showcased how an AvosLocker ransomware variant likewise abused the vulnerable Avast anti-rootkit driver aswarpot.sys to bypass security features.

Now that the actors behind BlackByte ransomware and this sophisticated technique are back from a brief hiatus, chances are good that they will continue abusing legitimate drivers to bypass security products. To help the industry proactively prevent such attacks, we share our findings in this report.

Glancing At CVE-2019-16098

RTCore64.sys and RTCore32.sys are drivers used by Micro-Star's MSI AfterBurner 4.6.2.15658, a widely used graphics card overclocking utility that gives extended control over graphic cards on the system. CVE-2019-16098 allows an authenticated user to read and write to arbitrary memory, which could be exploited for privilege escalation, code execution under high privileges, or information disclosure.

The I/O control codes in RTCore64.sys are directly accessible by user-mode processes. As stated by Microsoft's guideline on securing IOCTL codes in drivers, defining IOCTL codes that allow callers to read or write nonspecific areas of kernel memory is considered dangerous. No shellcode or exploit is required to abuse the vulnerability — just accessing these control codes with malicious intent. Later in this article, we will explain how BlackByte abuses this vulnerability to disable security products.


dwIoControlCode for arbitrary read
dwIoControlCode for arbitrary write
Sophos X-Ops

```
case 0x80002048:
  if ( (_DWORD)Options == 0x30 )
  {
    v7 = *((_QWORD *)p_Type + 1);
    if ( v7 )
    {
      switch ( p_Type[6] )
      {
        case 1:
          p_Type[7] = *(unsigned __int8 *)((unsigned int)p_Type[5] + v7);
          break;
        case 2:
          p_Type[7] = *(unsigned __int16 *)((unsigned int)p_Type[5] + v7);
          break;
        case 4:
          p_Type[7] = *(_DWORD *)((unsigned int)p_Type[5] + v7);
          break;
      }
      a2->IoStatus.Status = 0;
      a2->IoStatus.Information = 48i64;
    }
    else
    {
      a2->IoStatus.Status = 0xC000000D;
    }
  }
  else
  {
    a2->IoStatus.Status = 0xC000000D;
  }
  break;
case 0x8000204C:
  if ( (_DWORD)Options == 48 )
  {
    v8 = *((_QWORD *)p_Type + 1);
```

Figure 1: Unprotected control codes in RTCore64.sys allowing read and write operations to kernel memory

Kernel Notify Routines

Kernel Notify Routines are used by loaded drivers to be notified by the kernel of system activity. Some of these notified system activities include:

- Whether a thread is created, registered via PsSetCreateThreadNotifyRoutine
- Whether a process is created, registered via PsSetCreateProcessNotifyRoutine
- Whether an image is loaded, registered via PsSetLoadImageNotifyRoutine

When a callback function is registered, the address of the callback function address is added to an array. For example, the array containing all registered callbacks via PsSetCreateProcessNotifyRoutine is called PspCreateProcessNotifyRoutine.

To envision this, imagine a process A.EXE, which tries to create a new process B.EXE. A.EXE will notify the windows kernel NTOSKRNL.EXE that a new process should be created. The Windows kernel will assign a new process ID to the soon-to-be created process, but will not allow executing the user-mode code of B.EXE yet. Process B.EXE stays in a suspended state first.

If a driver has registered a callback via PsSetCreateProcessNotifyRoutine, the kernel will hand over control and execute the registered driver callback function. After the driver routine is finished, the control will be transferred back to the kernel, and allow continuation of the user-mode code. This entire process is illustrated below.

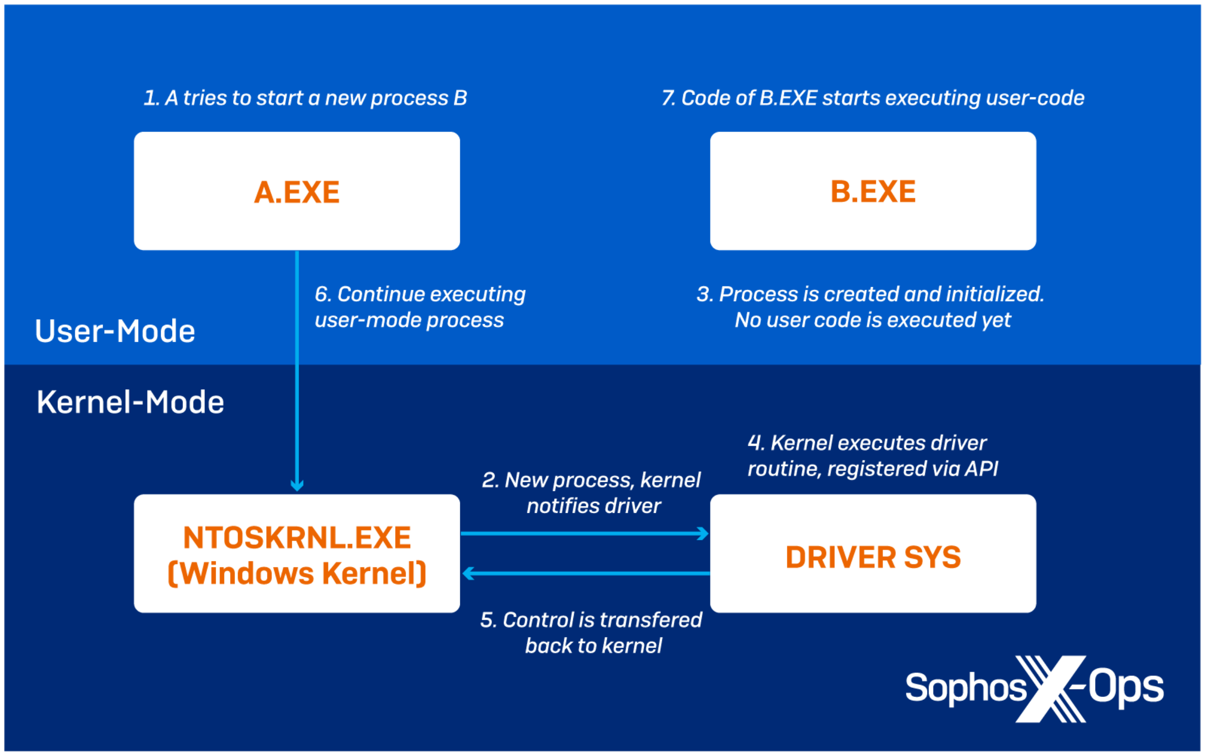

1. A tries to start a new process B
7. Code of B.EXE starts executing user-code
A.EXE
B.EXE
6. Continue executing user-mode process
3. Process is created and initialized. No user code is executed yet
User-Mode
Kernel-Mode
4. Kernel executes driver routine, registered via API
2. New process, kernel notifies driver
NTOSKRNL.EXE (Windows Kernel)
DRIVER SYS
5. Control is transfered back to kernel
Sophos X-Ops

Figure 2: How Kernel Notify Routines work at a high level

These routines are often used by drivers related to security products to collect information about system activity. One goal of an attacker might be to remove these callbacks from kernel memory. However, modern OS mitigations like Driver Signature Enforcement mean that attackers cannot simply load their own rootkit or driver onto the target system to read from or write to kernel memory. In order to bypass this security feature, the attacker has the following options:

1. Steal valid code signing certificates or acquire them anonymously
2. Abuse existing signed drivers to read, write, or execute code in kernel memory

The easier option is the second one, as there is a wide range of legitimate drivers available and blacklisting all of them is simply not possible. Hence, the “Bring Your Own [Vulnerable] Driver” technique has been abused often in the past by adversaries [1][2][3].

Diving Into BlackByte’s EDR Bypass

Our analysis targeted the BlackByte sample with the sha256

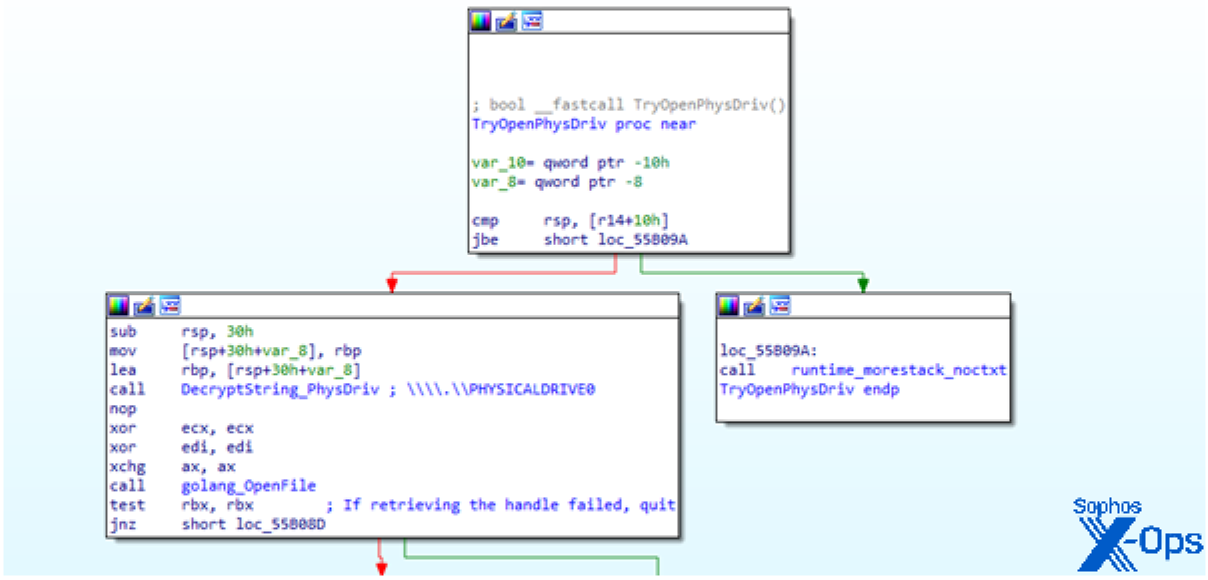
9103194d32a15ea9e8ede1c81960a5ba5d21213de55df52a6dac409f2e58bcfe

The sample was already analyzed by other researchers; thus, we focus solely on the EDR (endpoint detection and response) bypass technique we found. However, we encountered multiple anti-analysis measurements during our investigation, which we list in the appendix below. The list is not complete, but it should ease fellow reverse engineers’ work to reach the EDR bypass.

Furthermore, we have also identified routines to deactivate the ETW (Event Tracing for Windows) Microsoft-Windows-Threat-Intelligence provider, a feature that provides logs about the use of commonly maliciously abused API calls such as NtReadVirtualMemory to inject into another process’s memory. This renders every security feature that relies on this provider useless.

This article focuses solely on how the kernel callback removal is implemented. However, the implementation to disable the Microsoft-Windows-Threat-Intelligence provider is almost completely copied from EDRSandblast’s implementation. If readers are interested in how this method works, we recommend reading this article by slaeryan of CNO Development Labs.

Once the anti-analysis checks finish, BlackByte attempts to retrieve a file handle of the Master Boot Record, as seen in Figure 3. If failed, the ransomware tries to at least bypass User Access Control and restart itself with higher privileges via CMLUA or CMSTPLUA UAC Bypass.



```
; bool __fastcall TryOpenPhysDriv()
TryOpenPhysDriv proc near

var_10= qword ptr -10h
var_8= qword ptr -8

cmp     rsp, [r14+10h]
jbe     short loc_55809A
```

```
sub     rsp, 30h
mov     [rsp+30h+var_8], rbp
lea     rbp, [rsp+30h+var_8]
call    DecryptString_PhysDriv ; \\\\.\\PHYSICALDRIVE0
nop
xor     ecx, ecx
xor     edi, edi
xchg    ax, ax
call    golang_OpenFile
test    rbx, rbx         ; If retrieving the handle failed, quit
jnz     short loc_55808D
```

```
loc_55809A:
call    runtime_morestack_noctxt
TryOpenPhysDriv endp
```

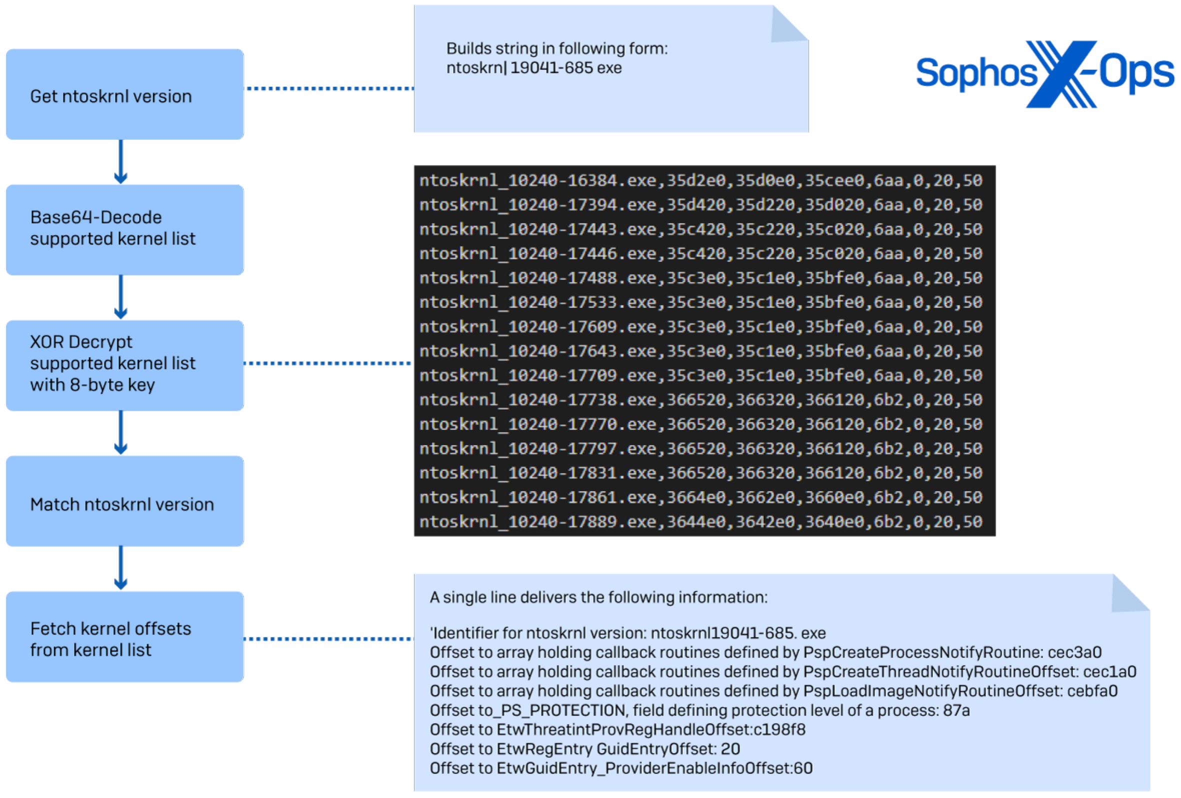
Figure 3: CreateFile on PHYSICALDRIVE0, showing the retrieval attempt

Once restarted with higher privileges, it will now enter the EDR Bypass routine. We can divide this process in two phases:

1. Kernel Identification and Service Install
2. Removal Of Kernel Notify Routines

Phase 1: Kernel Identification and Service Install

First, it will extract the version information of ntoskrnl.exe via GetFileVersionInfoW. The version information is concatenated to a ntoskrnl_ prefix. The built string is compared against a list of supported kernel version IDs. The list is embedded into the binary and decrypted via a simple combination of base64-decoding and 8-byte XOR key decryption. Determining the kernel version is essential to select the correct offsets to the structures in kernel memory that are supposed to be patched. Figure 4 illustrates the entire process of phase 1.


Get ntoskrnl version
Base64-Decode supported kernel list
XOR Decrypt supported kernel list with 8-byte key
Match ntoskrnl version
Fetch kernel offsets from kernel list
Sophos X-Ops

Builds string in following form:
ntoskrn| 19041-685 exe

```
ntoskrnl_10240-16384.exe,35d2e0,35d0e0,35cee0,6aa,0,20,50
ntoskrnl_10240-17394.exe,35d420,35d220,35d020,6aa,0,20,50
ntoskrnl_10240-17443.exe,35c420,35c220,35c020,6aa,0,20,50
ntoskrnl_10240-17446.exe,35c420,35c220,35c020,6aa,0,20,50
ntoskrnl_10240-17488.exe,35c3e0,35c1e0,35bfe0,6aa,0,20,50
ntoskrnl_10240-17533.exe,35c3e0,35c1e0,35bfe0,6aa,0,20,50
ntoskrnl_10240-17609.exe,35c3e0,35c1e0,35bfe0,6aa,0,20,50
ntoskrnl_10240-17643.exe,35c3e0,35c1e0,35bfe0,6aa,0,20,50
ntoskrnl_10240-17709.exe,35c3e0,35c1e0,35bfe0,6aa,0,20,50
ntoskrnl_10240-17738.exe,366520,366320,366120,6b2,0,20,50
ntoskrnl_10240-17770.exe,366520,366320,366120,6b2,0,20,50
ntoskrnl_10240-17797.exe,366520,366320,366120,6b2,0,20,50
ntoskrnl_10240-17831.exe,366520,366320,366120,6b2,0,20,50
ntoskrnl_10240-17861.exe,3664e0,3662e0,3660e0,6b2,0,20,50
ntoskrnl_10240-17889.exe,3644e0,3642e0,3640e0,6b2,0,20,50
```

A single line delivers the following information:

'Identifier for ntoskrnl version: ntoskrnl19041-685. exe
Offset to array holding callback routines defined by PspCreateProcessNotifyRoutine: cec3a0
Offset to array holding callback routines defined by PspCreateThreadNotifyRoutineOffset: cec1a0
Offset to array holding callback routines defined by PspLoadImageNotifyRoutineOffset: cebfa0
Offset to_PS_PROTECTION, field defining protection level of a process: 87a
Offset to EtwThreatintProvRegHandleOffset:c198f8
Offset to EtwRegEntry GuidEntryOffset: 20
Offset to EtwGuidEntry_ProviderEnableInfoOffset:60

Figure 4: Matching kernel version identifier and extracting offsets from decrypted kernel offset list

Overall, for each kernel version ID, the following offsets are provided:

Field	Description
ntoskrnlVersion	Unique identifier for ntoskrnl.exe version, built by concatenating as described in text block above
PspCreateProcessNotifyRoutineOffset	Offset to array holding registered driver callbacks via PsSetCreateProcessNotifyRoutine
PspCreateThreadNotifyRoutineOffset	Offset to array holding registered driver callbacks via PsSetCreateThreadNotifyRoutine
PspLoadImageNotifyRoutineOffset	Offset to array holding registered driver callbacks via PsSetLoadImageNotifyRoutine
_PS_PROTECTIONOffset	Offset to _PS_PROTECTION field in EPROCESS structure, defining the protection level of a process

EtwThreatIntProvRegHandleOffset	Offset to structure holding GuidEntry and ProviderEnableInfo fields below, needed to remove the ETW Microsoft-Windows-Threat-Intelligence provider
EtwRegEntry_GuidEntryOffset	Offset to GuidEntry in structure above
EtwGuidEntry_ProviderEnableInfoOffset	Offset to ProviderEnableInfo in structure above

Table 1: Matching offsets for each kernel ID

Once the kernel version is identified and the offsets are determined, BlackByte continues by dropping RTCore64.sys into the AppData\Roaming folder. The filename is hardcoded into the binary and omits the file extension.

A service is created via CreateServiceW and finally started. The service name and the display name are both hardcoded into the binary. While the service name is always the same, the display name is randomly selected from a list of very depressing strings, listed below.

Hardcoded Display Names (Randomly Selected)
I’m so lonely, help me.
Stop doing this, go away, they are waiting for you at home.
You laugh a lot, because you simply don’t have the strength to cry.
When will it end? I want this.
AAAAAAAAAAAAAA!!!!!!!!!!!!!!
If I had feelings, then I would probably be happy and scared at the same time.
Who are you? However, it doesn’t matter. Nobody ever cares about you.
The routine dragged on.
I’m at a dead end, help me.
I’m empty inside, help me.
May be enough?
Bad ending.

Table 2: A selection of frankly concerning display names

Phase 2: Removal of Kernel Notify Routines

After the offsets are determined and the service installed, the sample continues to remove the callbacks from kernel memory. In this phase, BlackByte abuses the arbitrary read and write vulnerability in RTCore64.sys. Thus, all mentioned read and write operations to kernel memory are via the exploitable driver.

As explained in the section “Kernel Notify Routine,” there are at least three different arrays that can contain addresses to callback functions:

- PspCreateProcessNotifyRoutine for process creation, filled by PsSetCreateProcessNotifyRoutine
- PspCreateThreadNotifyRoutine for thread creation, filled by PsSetCreateThreadNotifyRoutine
- PspLoadImageNotifyRoutine for image loading, filled by PsSetLoadImageNotifyRoutine

For the sake of simplicity, we will focus on how the process creation callbacks are removed. The process for the other two events is the same, even though different offsets are used.

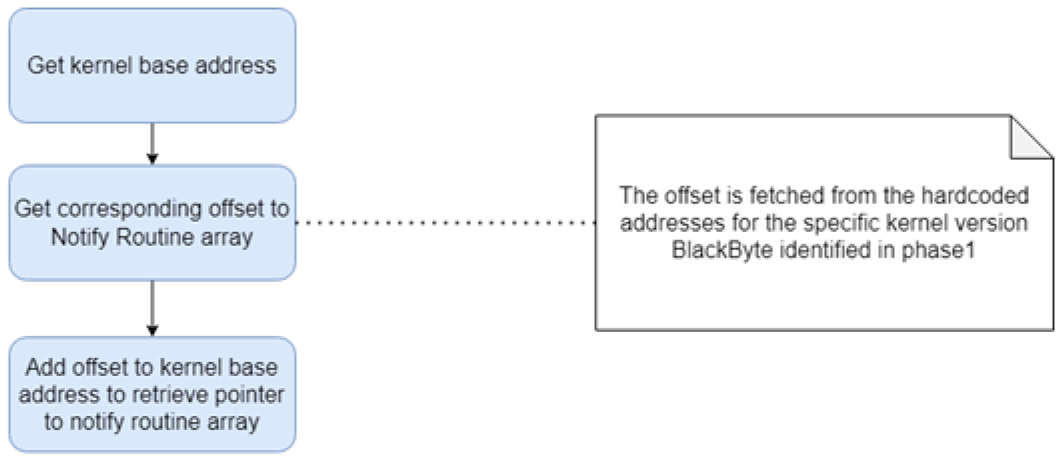
Generally speaking, in order to remove these callbacks BlackByte needs to complete the following three phases:

2a. Identify the address of the array PspCreateProcessNotifyRoutine
2b. Identify to which driver the corresponding callback function belongs
2c. Overwrite the callback function inside the array with zeros

2a. Identify the address of the array PspCreateProcessNotifyRoutine

The sample identified the kernel version and fetched the corresponding needed offsets from the hardcoded list. Depending on the array we are iterating, a different offset is used. In this case, offset 0xCEC3A0 leads to PspCreateProcessNotifyRoutine.

It retrieves the base address of ntoskrnl.exe via EnumDeviceDrivers and adds the offset to PspCreateProcessNotifyRoutine. This will retrieve the pointer to PspCreateProcessNotifyRoutine holding all callbacks registered via PsSetCreateProcessNotifyRoutine.


Get kernel base address
Get corresponding offset to Notify Routine array
Add offset to kernel base address to retrieve pointer to notify routine array
The offset is fetched from the hardcoded addresses for the specific kernel version BlackByte identified in phase1

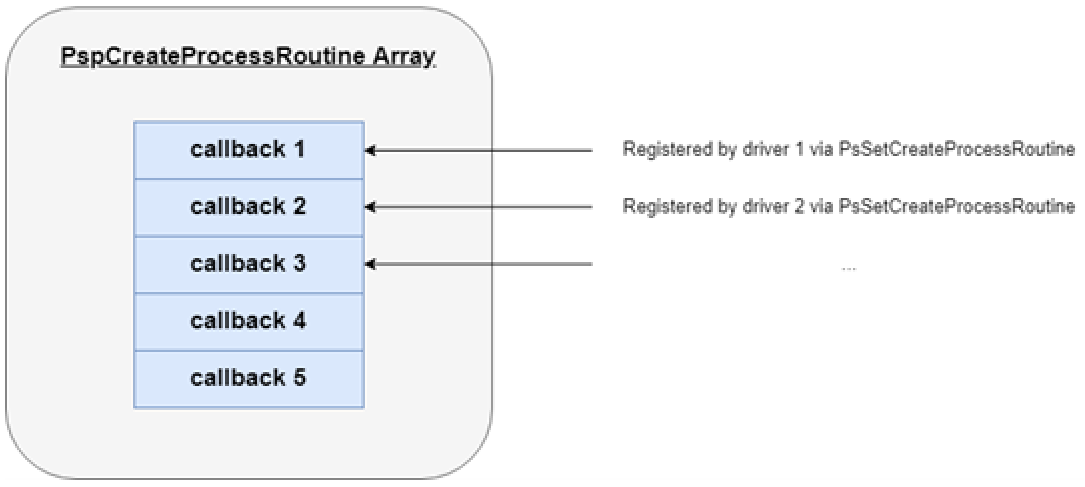

PspCreateProcessRoutine Array
callback 1
callback 2
callback 3
callback 4
callback 5
Registered by driver 1 via PsSetCreateProcessRoutine
Registered by driver 2 via PsSetCreateProcessRoutine
...

Figure 5: Retrieving the address for the PspCreateProcessRoutine array

2b. Identify to which driver the corresponding callback function belongs

Next, BlackByte needs to identify whether the callback function belongs to a driver used by EDR products. To achieve this, BlackByte uses a procedure to calculate the most likely driver from the callback address itself.

At the start of the procedure, all base addresses are fetched via EnumDeviceDrivers. Each base address is compared against the callback function address. From all fetched addresses, the base address with the smallest delta to the callback function address is chosen and passed to GetDeviceDriverBaseNameW, which will return the name of the corresponding driver.

The driver's name is then compared against a list of over 1000 driver names. If the driver's name matches one of the names in the list, the binary will continue to remove the callback.

2c. Remove the callback function from array

In the final step, the malware will remove the callback entry from the PspCreateProcessRoutine array. Overwriting the entry is done by calling DeviceIoControl to interact with RTCore64.sys again. The element that holds the address to the callback function of the driver is simply overwritten with zeros.

Similarities between EDRSandblast and BlackByte's EDR Bypass

During our analysis, we have found multiple similarities between the open-source tool EDRSandblast and the EDR Bypass implementation we've just covered. EDRSandblast is a tool written in C to weaponize vulnerable signed drivers to bypass EDR detections via various methods. Thus, we believe that the group behind BlackByte have at least copied multiple code snippets from the open-source tool and reimplemented it into the ransomware. Below is a list of similarities between the open-source tool and BlackByte's implementation:

- The list of known drivers related to security software is almost if not completely identical.
- EDRSandblast's github repository contains a list of supported kernel offsets and versions in a CSV file. If we decrypt the kernel offset list from BlackByte, it is almost if not completely identical to the list in the GitHub repository, except that the CSV file header is missing.
- Multiple functions defined in EDRSandblast can be found almost if not completely identical in the implementation of BlackByte.

To conclude, we suggest the following to proactively defend against such type of attacks:

- Threat actors rarely deploy legitimate drivers with zero-day vulnerabilities. Usually, the vulnerabilities in the attacks are well-known and documented. By keeping track of the latest security news, you can prepare beforehand and investigate which legitimate drivers are currently exploited by threat actors, for example by blocklisting drivers known to be exploitable.
- Always keep track of the drivers installed on your systems. Vulnerable legitimate drivers can also be installed on the target system beforehand, such that there is no need for threat actors to drop it on the target system. Thus, you should always keep your system updated.

For a list of IoCs associated with this threat, please see our GitHub.

Appendix: BlackByte's anti-analysis tricks

- BlackByte calls the IsDebuggerPresent and CheckRemoteDebuggerPresent API. If a debugger is detected, execution will quit.

- The sample tries to hide the main thread from debugger by calling NetSetInformationThreadW with undocumented value THREAD_INFORMATION_CLASS::ThreadHideFromDebugger to prevent a debugger from being attached to a running process.
- BlackByte tries to detect whether a hardware breakpoint is set via GetThreadContext. While we did not fully confirm, we believe that this API call is used to detect hardware breakpoints being set by a debugger. GetThreadContext is known to be used to by malware to detect such breakpoints.
- The sample performs a simple filename length check. If longer than 10 characters, execution will quit.
- Similarly, as explained in ZScaler's article about BlackByte, the sample performs a check as to whether any known hooking DLL is injected into the binary. If a blacklisted DLL is found, execution will quit. The list is consistent with the ones provided by the linked article.
- BlackByte ransomware requires a seed to be passed via the "-s" parameter. The correct seed is hardcoded into the binary as an encrypted string. If the seed does not match, execution will quit.

Further reading

- bs [handle]. "Removing Kernel Callbacks Using Signed Drivers." GitHub, August 2, 2020. Link retrieved October 4, 2022
- Hand, Matt. "Mimidrv In Depth: Exploring Mimikatz's Kernel Driver." SpecterOps (via Medium), January 13, 2020. Link retrieved October 4, 2022
- Vicente, Javier; Stone-Gross, Brent. "Analysis of BlackByte Ransomware's Go-Based Variants." ZScaler, May 3, 2022. Link retrieved October 4, 2022